ANALYSIS OF JAVA PROGRAMS USING BYTECODE BASED FRAMEWORKS

Ranjan Kumar

Roll. 213CS3185

under the supervision of

Prof. Durga Prasad Mohapatra

Department of C.S.E

NIT Rourkela

Rourkela – 769008, India

ANALYSIS OF JAVA PROGRAMS USING BYTECODE BASED FRAMEWORKS

Dissertation submitted in

MAY 2015

to the department of

Computer Science and Engineering

of

NIT Rourkela

in partial fulfillment of the requirements

for the degree of

Master of Technology

by

Ranjan Kumar

(Roll. 213CS3185)

under the supervision of

Prof. Durga Prasad Mohapatra

Department of C.S.E

NIT Rourkela

Rourkela – 769008, India

Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela-769 008, India. www.nitrkl.ac.in

May 26, 2015

Certificate

This is to certify that the work in the thesis entitled *ANALYSIS OF JAVA PROGRAMS USING BYTECODE BASED FRAMEWORKS* by *Ranjan Kumar*, having roll number 213CS3185, is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of *Master of Technology* in *Computer Science and Engineering Department*. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Dr. Durga Prasad Mohapatra

Associate Professor

Department of CSE

NIT, Rourkela

Acknowledgment

First of all, I would like to express my profound feeling of respect and appreciation towards my supervisor Prof.Durga Prasad Mohapatra, who has played a pivotal role in successful completion of my thesis work. I want to express gratitude toward him for acquainting me with the field of Program Slicing and giving me the chance to work under him. His unified confidence in this area and capacity to draw out the best of systematic and practical abilities in individuals has been pivotal in intense periods. Without his pivotal guidance and help it would not have been possible for me to finish this thesis. I am significantly obliged to him for his consistent consolation and important guidance in every part of my academic life. I think of it as my favorable luck to have got a chance to work with such a radiant person.

I thank our H.O.D. Prof. S.K Rath for their constant support in my thesis work. They have been extraordinary wellsprings of motivation to me and I say thanks to them from the bottom of my heart.

I would also like to thank PhD researchers Subhrakanta Panda, Jagannath Singh and my friends Raj, Ravi and Rohan to give me their regular suggestions and supportive gestures during the entire work.

At last but not the least I am in the debt to my family to support me regularly during my harsh times.

I wish to thank all faculty members and secretarial staff of the CSE Department for their sympathetic cooperation.

Ranjan Kumar

Abstract

Java SDG(System dependence Graph) API and JOANA (Java Object-sensitive Analysis) are two bytecode based analysis frameworks available for analyzing object oriented java programs for different applications. In the present era, the continuous evolution of the customer expectations and requirements has resulted in the increase of size of the software. This arises the problems in maintaining software. Both the frameworks i.e Java SDG API and Joana consist of different variety of analysis techniques which are based on dependence graph generation and computation of slices of an input program. In our work, we make a comparative analysis study on the effectiveness and efficiency of both these above mentioned analysis frameworks in generating the corresponding intermediate dependence graph and computing slices. The dependence graph we have generated is SDG and we have used backward slicing approach in order to compute slices. The two-phase graph reachability algorithm is used in our work in case of Java SDG API in order to perform slicing. The two web start applications are used in order to generate and view SDG in case of Joana which are IFC console and Joana graph viewer. The analysis is based on the bytecode of the program under consideration. The experimental analysis shows that Joana can be extended for more diverse applications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the present era, efficiency of software is a key factor for which everybody is looking for. So in our work, we review the Java programs via two frameworks i.e Java SDG API and Joana. A well-known data structure in order to analyse programs is dependence graph [2]. Testing, merging, understanding, debugging of programs are few applications of program slicing [5,14]. The dependence relation among statements should be known apriori for computing correct and precise slices of the input program. Once the dependences among the program components are known, the program is then graphically represented. Many such intermediate graphical representations [1,3,4] have been introduced yet for effective program comprehension and analysis.

Program Dependence Graph (PDG) [2] is a graphical representation of the program under consideration. The different edges correspond to the dependency among statements and vertices correspond to the statements which are present in the program. SDG (System Dependence Graph) is a set of PDG. In order to handle complex programs that consists of multiple procedures, PDG is not suitable. So, in order to deal with this problem PDG is extended to SDG.

In our work, we analyze the effectiveness and efficiency of two frameworks i.e. Java SDG API [5] and JOANA (Java Object-sensitive Analysis) [3], by generating the dependence graph of the input program. The generated SDG is then used to

compute the slices with respect to some point of curiosity known as slicing criterion. As both the frameworks under consideration are bytecode based, we compute the slices taking the bytecode of the program as input.

1.1 Motivation

The type of software which we use in this present era is very large in size and also complex in nature which makes us difficult to understand, maintain, test and debug the code. In a program for locating the bugs, we search the entire program statement by statement which is a tough and more time consuming task. In order to deal with these issues, Weiser introduced an approach i.e. program slicing that helps in finding the interdependence statements contained in the program. The interdependence statements are the statements which shows the dependencies between the statements of two different procedures. The slicing algorithms proposed until now by the different researchers deals with Object oriented programming and some of them deals with Aspect oriented programming [10]. They have taken SDG for representing intermediate graph in order to compute slices but it is not assured clearly about the generation of SDG and also not stated that which takes more time in the process of computing slice.

1.2 Objectives

The major objectives of our work are:

- To construct an intermediate representation of Java programs known as SDG with the help of two different frameworks i.e. Joana and Java SDG API.
- To compute the slice using the SDG generated by Joana and Java SDG API.
- To perform a comparative analysis between the results obtained for JOANA and Java SDG API based on the input programs.

1.3 Organization of the Thesis

The thesis comprises of the below given chapters:

1. Chapter 1: This chapter comprises of the introduction part in which we discuss about the two frameworks i.e. Joana and Java SDG API and also discuss the program slicing concept. This chapter also includes motivation and objective of our research work.

2. Chapter 2: This chapter shows the fundamental concepts which are useful and related to our work.

3. Chapter 3: This chapter presents the literature review where we have explained some existing works on SDG generation and program slicing.

4. Chapter 4: In this chapter, we show how the SDG is generated using the two frameworks i.e Joana and Java SDG API and also compute the slices from the generated SDG.

5. Chapter 5: In this chapter, we show a comparative analysis between the above mentioned two frameworks.

6. Chapter 6: In this chapter, we conclude our research work.

Chapter 2

Fundamental concepts

In this chapter, we will discuss some basic concepts of program slicing which will help to understand details of this chapter. Again we will discuss the concepts of Dependence graphs and Graphviz.

2.1 Program Slice

A Set of statements that produces an effect on the value of a variable in a given statement s is a program slice [14]. In this chapter, we will discuss the fundamental concepts which are related to our work. In this section, we will discuss about different types of program slicing.

2.1.1 Static slicing

In this type of slicing, the dependency among statements of the program is taken into consideration for every conceivable input data.

2.1.2 Dynamic slicing

In this type of slicing [11], the dependency among statements of the program is taken into consideration for a particular input data. It also helps to decrease the size of

imprecise computation of static slice.

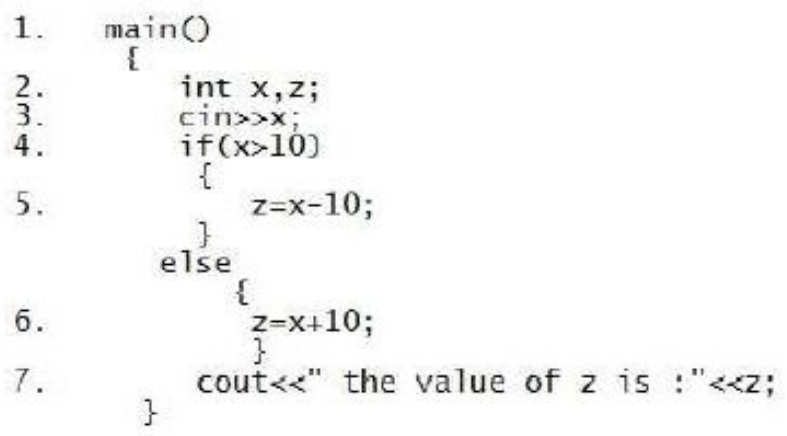

Figure 2.1: An example program for Static and Dynamic Slicing.

In static slicing, slice with slicing criterion <7,x>contains line numbers 1,2,3,4,5,6,7. But in dynamic slicing, slice with slicing criterion <7,x>contains line numbers 1,2,3,4,6,7.

Graph traversal based slicing techniques are:

2.1.3 Forward slicing

The subset of program statements under consideration, that might be influenced by the variable of interest at the statement taken into consideration is a forward slice [7].

2.1.4 Backward slicing

The subset of program statements under consideration, that might have influenced the variable of interest at the statement taken into consideration is a backward slice [7].

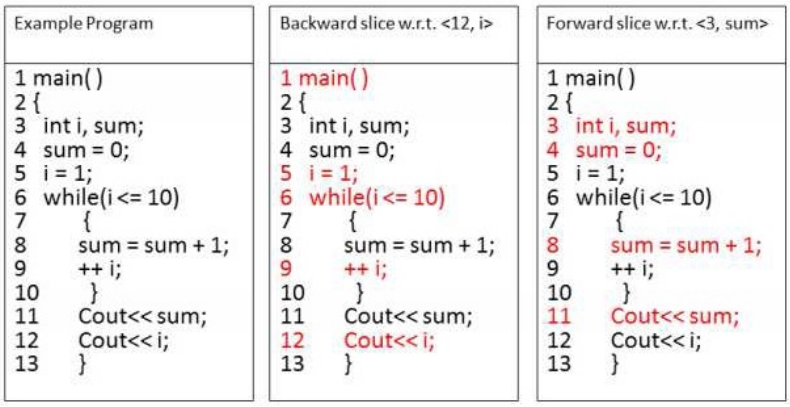

Figure 2.2: Backward and forward slicing.

2.2 Dependence Edges

Dependence edges correspond to the various dependences existing between the statements of a program. In this thesis, we emphasize on two different kinds of dependencies as given below:

2.2.1 Control dependence edge

Control dependence edge [10] is an edge that corresponds to the relationship among the two operations. These two operations has the functionality that one executes after the other. Also, it specifies the execution order of these operations.

2.2.2 Data dependence edge

An edge that corresponds to the relationship among the two statements where the computational outcome produced by one statement is used by another statement is data dependence edge [10]. The given below example shows the data dependence and control dependence:

From the above example, statement s6 is data dependent on the statement s3 and statemnet s7 is control dependent on statement s5.

```
s1.    main ( )
       {
s2.          int  x, y,z;
s3.          x=10;
s4.          y=20;
s5.          If(x>y)
s6.                cout<<" the largest value is"<< x;
           else
s7.                cout<<" the largest value is"<< y;
       }
```

Figure 2.3: Example of Data dependence and Control dependence.

2.3 Dependence graph

The different statements of the program and the dependences among them are modeled graphically to form the intermediate dependence graph. The two widely used dependence graphs in the existing literature on program slicing are discussed below:

2.3.1 Program dependence graph

A PDG is a graphical representation of a method in a program. This dependence graph is a pivotal component in the process of generating SDG. Its edges represent the control predicates and dependency among statements and nodes represents the statements that builds the program.

2.3.2 System dependence graph

After looking at a complex program, we can say that a program does not only consists of one single method, but comprises of several number of methods. For such type of cases, a PDG is extended to SDG as PDG is not enough to represent the entire information regarding the program. SDG helps us in producing more

precise slices from programs containing multiple procedures, because it contains the information about actual procedures calling context. SDG is basically a collection of PDGs. SDG consists of five new vertices more than PDG: Actual-in vertex, Formal-in vertex, Actual-out vertex, Formal-out vertex and Call-site vertex. SDG also comprises of three new kinds of edges which are:

- Parameter-in edge: Parameter-in edges are the edges that are added from actual-in nodes at a call-site to the corresponding formal-in nodes in the called procedure.
- Call edge: A call edge is an edge that is added from each call-site to the entry node of the called procedure.
- Parameter-out edge: Parameter-out edges are the edges that are added from formal-out nodes of each procedure to corresponding actual-out nodes at each call-site.

2.4 Graphviz

Graphviz (Graph visualisation software) [17] is a package of open source tools developed at AT&T labs Research for drawing graphs. Graphviz is an open source software licensed under the Eclipse Public License. DOT language scripts are used to read the contents of the graphviz file. It provides libraries for different software applications to use along with other tools.

2.5 Application of Program Slicing

Program slicing has many applications which we have discussed below. Initially, program slicing concept is used to develop automated code decomposition tools. Program debugging is the primary objective behind the development of these tools.

The program slicing techniques has many applications in the field of software development process.

2.5.1 Differencing the programs

Basically, software engineers discover difficulties to differentiate two programs. So, program slicing technology can be utilized efficiently for differentiating two programs. It makes a difference to discover all the parts of distinctive programs having diverse conduct and to create a program that catches the semantic contrasts between two programs by contrasting the backward slices of the vertices in two dependence graphs. Here, the backward slice is computed with the help of slicing criterion.

2.5.2 Software Maintenance

Software maintenance [15] is an expensive procedure due to the fact that every change to a program source code must consider into numerous unpredictable dependence relationships in the current programming. The most difficult part in the software maintenance, is to comprehend different dependencies in the available software and to make alterations to the currently available software without presenting new issues, i.e. whether a code change in a program will make any influence to the conduct of different codes of the program. In order to solve this issue, it is pivotal to know which variables will be relied on upon which statements. This issue can be diminished when the software will go through slicing technique concept.

2.5.3 Refactoring

Basically, refactoring [15] is characterized as the procedure of enhancing the configuration of currently available software frameworks. In such a situation, there is a change in source code happens. At the time of changing, every transformation is

relied upon to save the conduct of framework. There is straightforward illustration of refactoring is removing a procedure from one class to another. Henceforth for the instance of refactoring, program slicing plays a pivotal part as it discovers the subset of statements of a program which affect the value of a variable in a given statement s.

2.5.4 Debugging

Debugging [15] helps in discovering and minimizing the number of defects in the project. The process of discovering defects in a system is a troublesome task. The procedure to discover a defect includes running the program many number of times which is more time consuming task because we have to search each line. In distributed system, this issue is more troublesome in view of different dependencies i.e. control dependencies, data dependencies furthermore communication dependencies that may discover extra defects. Program slicing was initially proposed for looking at the procedure of debugging done by software engineers. Software engineers virtually compute slice while debugging codes which was troublesome and time consuming. That's why, program slicing methods makes a difference to discover the subset of explanations as indicated by their dependencies from which it is simple to discover bugs in an efficient way.

2.5.5 Functional Cohesion

Cohesion [15] has the functionality to measure the degree to which the component of a module belong to each other. When there is no further chance of division of module into sub-module then the software is said to be highly cohesive. The cohesion should be high in order to achieve a good quality software. We need program slicing concept in order to get the interdependence statements within a program.

2.5.6 Testing

Testing [15] is basically used for finding the errors existing in software or a program. In order to maintain software, there is a frequent use of regression testing. As we know that errors occur while testing the software, we use regression testing to re-test the software after the modifications. After making either a little change to the code of the software, many tests are required in order to check that no more unwanted behaviour arises due to that little change. So, new test cases are required along with the previous test cases. For deducting the number of test cases, we use the slicing concept.

2.6 Summary

This chapter explains about the program slicing concept and its various types. We have also discussed about types of dependence edges and dependence graphs. We have explained about the graphviz software which we have used further in order to show the generated SDG. There is also an explanation of usefulness of program slicing by describing its application.

Chapter 3

Literature Review

In this chapter, we explain the survey of some existing papers which are correlated to our work.

Weiser [6] is the one who proposed the first program slicing approach for procedure oriented programs. According to Weiser, program slicing is a method of decomposition that helps in extracting the statements from programs, those are pertinent for a particular computation. Program slicing is a new means of decomposing the programs automatically. Limited to code previously written, it may prove helpful during the testing, debugging, and maintenance of the software.

Horwitz et al. [18] introduced a SDG (System dependence graph) as an intermediate representation for the programs which include multiple procedures. They proposed two-phase graph reachability algorithms to compute the slices. Larsen and Harold extended the SDG of Horwitz et al. [18] for representing object oriented programs. They include many object oriented features such as class, objects, inheritance, polymorphism etc on System Dependence graph.

Wang et al. [1] introduced a method that proceeds by backward traversal of the byte code traces produced by an input I in a given program p.

Liang et al. [10] presented the generation of SDG with the help of object-oriented programs. They presented an approach which is more accurate and efficient to construct than existing approaches. They represented the SDG in such a way that

it supports precise slicing than other approaches. The generated SDG recognizes data members which belong to different objects. In the case, when objects are used as parameters their approach represents data members and also represents the impacts of polymorphism on parameters and callsites. They have also presented a concept of object slicing which helps the user to examine the impact of an object on the slicing criterion.

Silva et al. [7] surveyed the existing work on program slicing-based techniques. He described each individual technique by elaborating its characteristics and main applications. He also showed an example of slicing by using each individual technique. Each one of the slicing techniques is compared in order to get the detailed information about the relations between them.

Walkinshaw et al. [2] introduced the concept of System Dependence Graph consisting of muti-procedures. They presented a Java System Dependence Graph which provides better speed and precision than conventional methods. They represented object classes and interfaces in order to treat objects and object data members of any operation individually.

Chapter 4

Generation of SDG and Slice Computation

In chapter 4, we explain the process of SDG generation and slice computation via two frameworks: Joana and Java SDG API, for analyzing the programs under consideration.

4.1 Block Diagram of our Approach

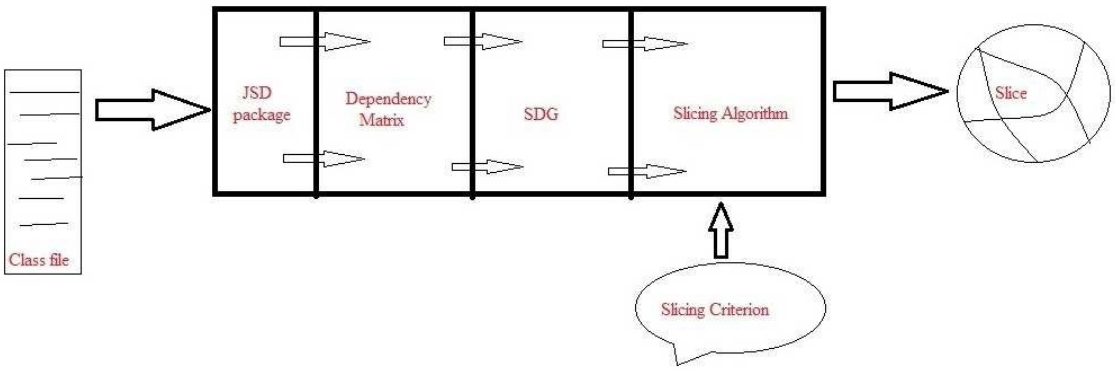

Figure 4.1: Block Diagram of our approach.

The above figure shows actually how our tool works. First we create the class

file of the Java program to be sliced. Then we have to give that class file to our tool. After that it finds all the dependence matrix using Java system dependence package (JSD package). From that matrix it creates the SDG of that Java program. After generating SDG, in order to compute slice of the specified Java program we have to give a slicing criterion and then apply a slicing algorithm.

4.2 Creation of SDG of Java programs

A Java System Dependence Graph is a multi-graph which contains control and data dependencies between the statements of a Java program. It contains classes, methods, statements, interfaces to represent SDG of Java program. Each of these represent graph separately and combine with hierarchical manner to make complete SDG of Java program. Here, the statements are lower level then method level like this all are connected in a hierarchical structure within the SDG. Now, we discuss the different steps to create SDG.

4.2.1 Statement dependency Graph

Statements are the lowest level in SDG of Java program. A statement is basically an atomic construct which represents a single expression in the program source code. In order to represent a call to another method (a callsite), there is a requirement of a special type of representation .

4.2.2 Method dependency Graph

It is used to represent a single method or procedure of a program. The method entry vertex connects to other members of methods using control dependence edges. Parameter passing is obtained by using actual and formal vertices. The called procedure has formal-in and formal-out vertices, which use parameter variables accordingly. There is a call dependence edge which connects between the call site

and the procedure being called.

4.2.3 Class dependency Graph

It represents the classes of the program. The next layer to method dependency graph is class dependency graph. It contains class entry vertex to connect the method entry vertices by using class member edges. Here, dependent classes are connected by using class dependence edges.

4.2.4 Construct the JSDG

Here, we have taken one class named as JavaSDG to find all the information regarding different dependence as discussed previously. This class contains different linked list for storing different nodes and the dependencies between them. There is a class named as ConvertJsdgToGv which converts Graph using all the information from stored matrix. Finally, we give a specific path to store the SDG of input program.

4.3 Computing slices using Java SDG API

SDG API helps in building a program slicing tool. This program slicing tool helps in reducing the price invested during each cycle of software development and its maintenance. SDG API also detects similar codes within the source code of a project, therefore it is also an important issue to prevent the occurrence of similar problems.

Algorithm for slicing (two-pass graph reachability algorithm)

In our work, we employ the two-phase graph reachability algorithm proposed by Horowitz et al. [17, 9] to compute the slices. The slicing algorithm basically consists of two passes:

Function	Description
ControlDependenceBFS	Breath first search following the control dependence edge.
ControlDependenceDFS	Depth first search following the control dependence edge.
DataDependenceDFS	Depth first search following the data dependence edge.
NodeCount	Return the number of nodes in a SDG.
PDGCount	Return the number of PDGs in a SDG.
PDGRetrieve	Return a PDG by a call node.

Figure 4.2: Functions in SDG API.

- **Pass 1**: The algorithm has the functionality of traversing backward along all the edges except parameter-out edges, and marks all the vertices reached during the traversal.
- **Pass 2**: The algorithm performs backward traversal from all the vertices marked in the first pass. It traverses along all the edges except call and parameter-in edges, and marks the vertices reached during the traversal.
- The final slice is given by the union of all the vertices marked in Pass 1 and Pass 2.

Figure 4.2 shows SDG API functionalities which helps in creating SDG [5]. One of the best way to know about ASM is to write a Java source file which is equivalent to what you want to generate and after that for seeing the equivalent ASM code use the ASMifier mode of bytecode plugin for eclipse. In Java SDG API, we utilize bytecode based ASM framework [8] for analyzing and manipulating the bytecode. ASM can be used in order to change existing classes or to dynamically create the classes straightforwardly in binary form. The available frequent transformations and analysis algorithms in ASM framework permit to effortlessly assemble customized complex transformations and code analysis tools. A sample program given in Figure 4.3 is used to generate the SDG. The input program with respect to the slicing criterion <19,z >is shown in Figure 4.5.

ASM offers the same functionality as other bytecode frameworks, but its major goal is to focus on performance parameter and its ease of use. In order to do so, it

```
public class FindLargest {
     public static void main(String args[])
       {
          int x, y, z;
          System.out.println("Enter three integers ");
          Scanner in = new Scanner(System.in);

          x = in.nextInt();
          y = in.nextInt();
          z = in.nextInt();

          if ( x > y && x > z )
             System.out.println("x is largest.");
          else if ( y > x && y > z )
             System.out.println("y is largest.");
          else if ( z > x && z > y )
             System.out.println("z is largest.");
          else
             System.out.println("Entered numbers are not distinct.");
       }
}
```

Figure 4.3: A sample source program.

should be made as small and as fast as possible. Due to this feature, it can be used in dynamic systems. ASM is a name in itself as it has no full form.

Figure 4.4 shows the SDG of the sample program given in Figure 4.3. The generated SDG of the sample program is visualized using graph vizualization tool. In our work, the class file of the sample program as an input for generating the corresponding SDG is considered. ASM framework is used here for analyzing and manipulating the Java bytecode. Figure 4.5 shows the slicing criterion given to compute the required slice with respect to the variable of interest. We have considered node number 17 in order to compute the slice required. The node number 17 corresponds to statement number 19 in the sample program shown in Figure 4.3. The node number in the SDG generated is decided by reading the input program from top to down and by looking at the SDG generated corresponding to each statement of the input program. We have generated a sliced SDG after performing a backward slicing concept on node number 17 which is depicted in Figure 4.6. In Figure 4.6, the solid lines represent a control dependence edge and the dotted lines

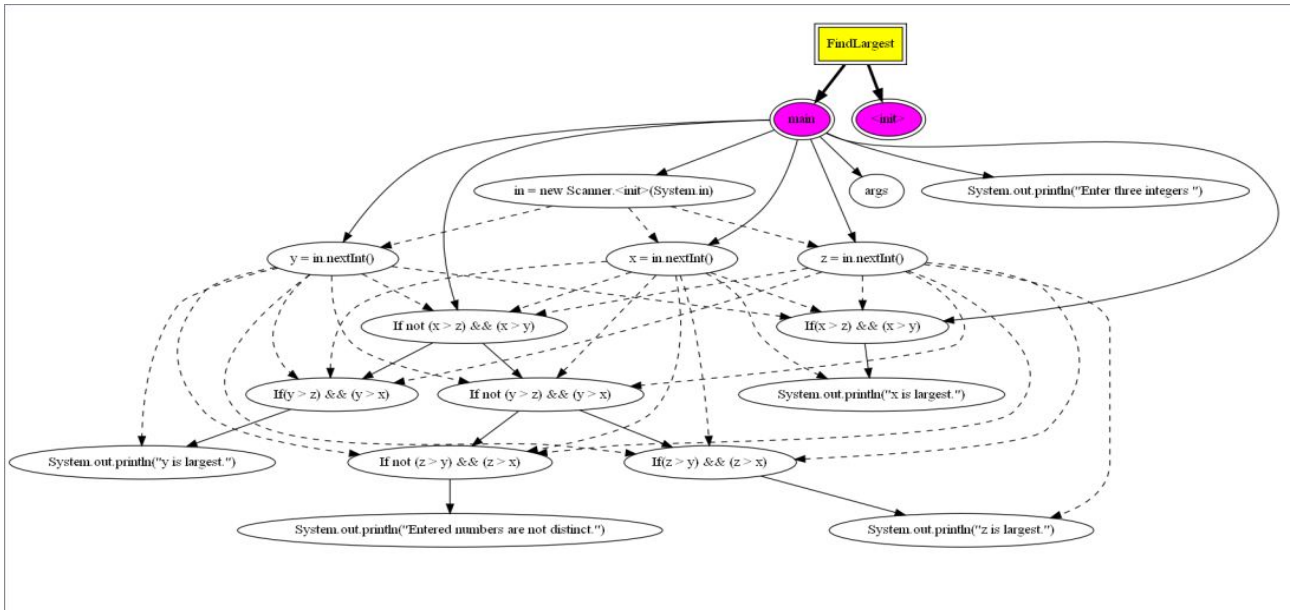

Figure 4.4: SDG of the sample program shown in Figure 4.3.

represent data dependence edge.

4.4 Computing slices using Joana

Joana is full bytecode based analysis framework for Java language. It builds a system dependence graph (SDG) by taking source code of the program as input. This graph corresponds to the information flow among the statements present inside the program. SDG contains nodes that correspond to each statement of the program. While the edges correspsond to the information flow between these nodes.

The edges of the SDG generated by Joana framework represent both the direct dependencies through values called data dependencies and also indirect dependencies known as control dependencies. When the outcome of a statement decides whether another statement is to be executed or not then there comes the concept of control dependencies. For example, the condition of an while-statement judge the execution of the next statement. For creating a SDG using Joana, we need two web start applications. The two web start applications are IFC console and Graph viewer [3].

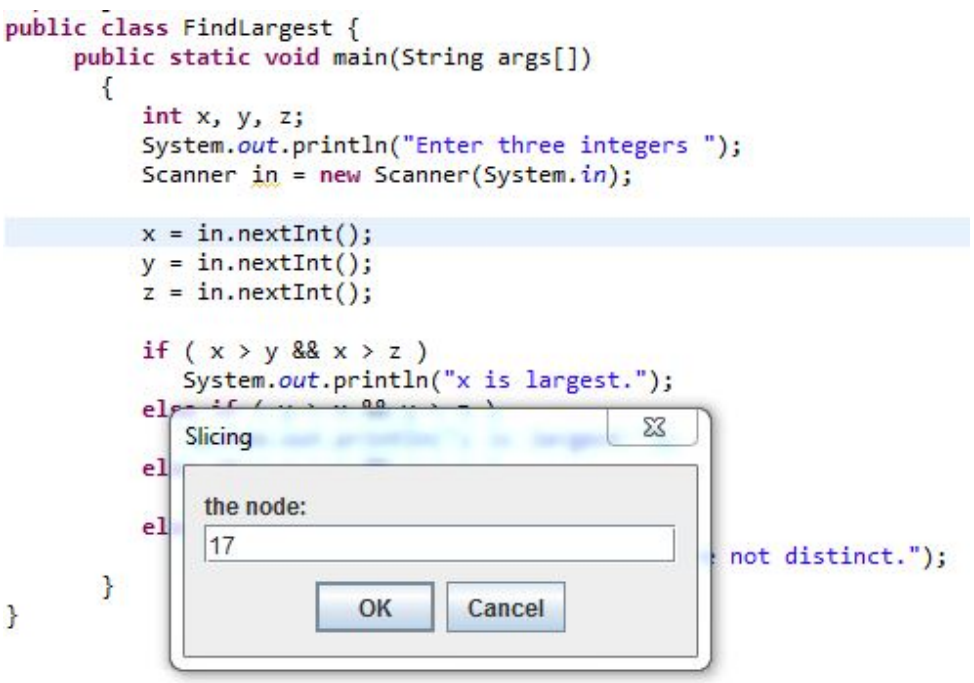

Figure 4.5: Input program with respect to slicing criteria <19, z >.

IFC (information flow control) console has a number of applications. One of them is the generation of system dependence graph. IFC console is a GUI (graphical user interface) that hides the majority of JOANAs internal characteristics. The SDG generated by the IFC console is viewed by an application i.e. Graph viewer. Graph viewer has also the feature of computing slices and chopping on the generated SDG.

Another source program as given in Figure 4.12 is taken to demonstrate the SDG generated by Joana. The SDG shown in Figure 4.13 contains five more new types of vertices as compared to PDG:

- Call site Vertex: Call vertices represent a call site vertex in a method.
- Actual-in vertex: This vertex shows the flow of actual parameters to call temporaries.
- Actual out: This vertex represents the flow of actual parameters from return temporaries.

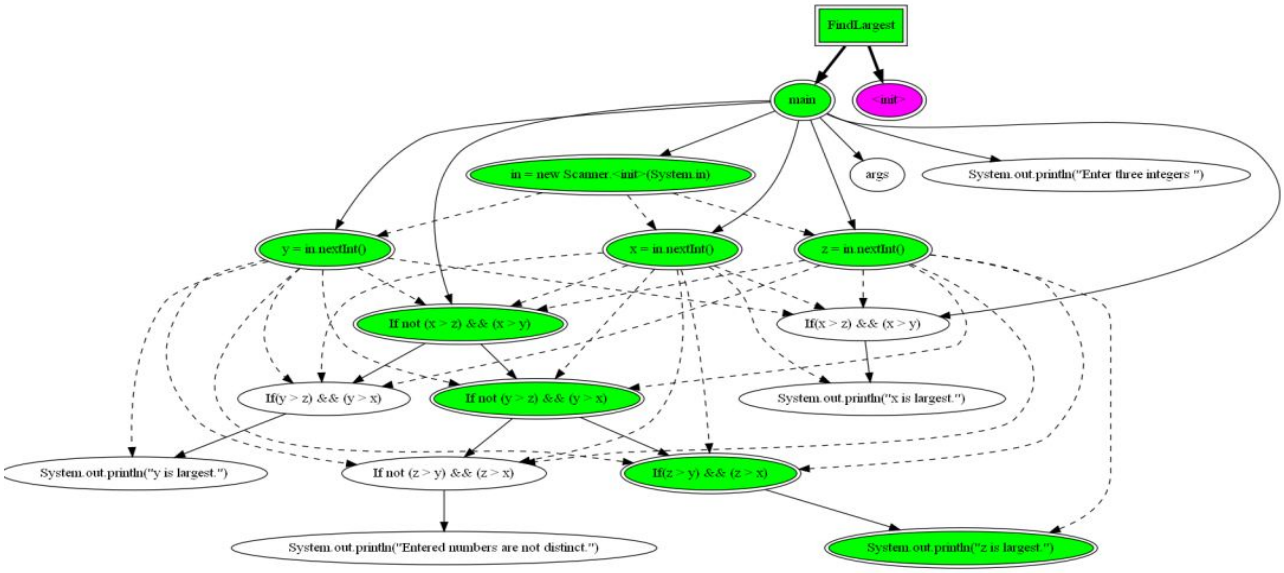

Figure 4.6: System Dependence Graph showing the slices w.r.t slicing criterian <19, z>.

- Formal-in vertex: This vertex is the callee analogs of actual-in vertex.
- Formal-out: This vertex is the callee analogs of actual-out vertex.

Formal-in parameters receives the values from call sites and formal-out parameters receives the return values. Formal-in vertex is control dependent on the entry node of the called procedure during par-in edge representation and formal-out vertex is control dependent on the entry node of the procedure during the representation of par-out edge. The statements call can be made in two ways either from the actual parameters to call temporaries or from return temporaries to actual parameters. This explicit modeling of procedure invocation restricts dependences between procedures to dependences between actual-in vertices to formal-in vertices and from formal-out vertices to actual-out vertices. The Figure 4.14 represents a sliced system dependence graph for the sample program given in Figure 4.12. We have taken node number 7 as slicing criterion for computing the slice.

The time required for SDG generation is shown in Figure 4.15. Joana takes

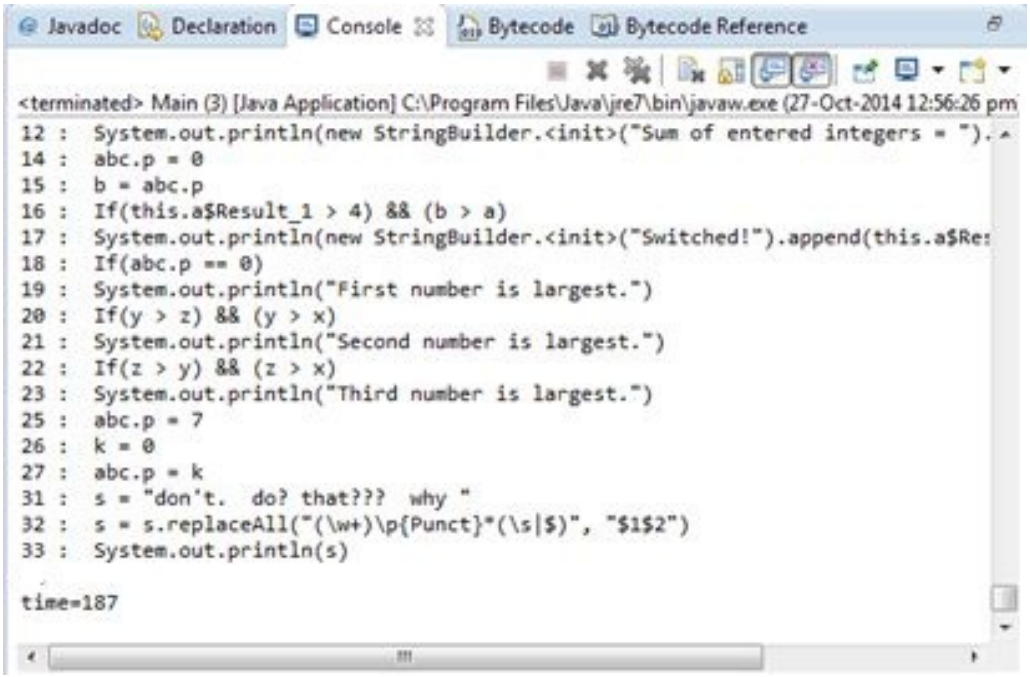

Figure 4.7: Time required for SDG generation (t1)=187ms.

166ms in order to generate SDG and also it calculates the space required by SDG which is 50M for the program shown in Figure 4.12.

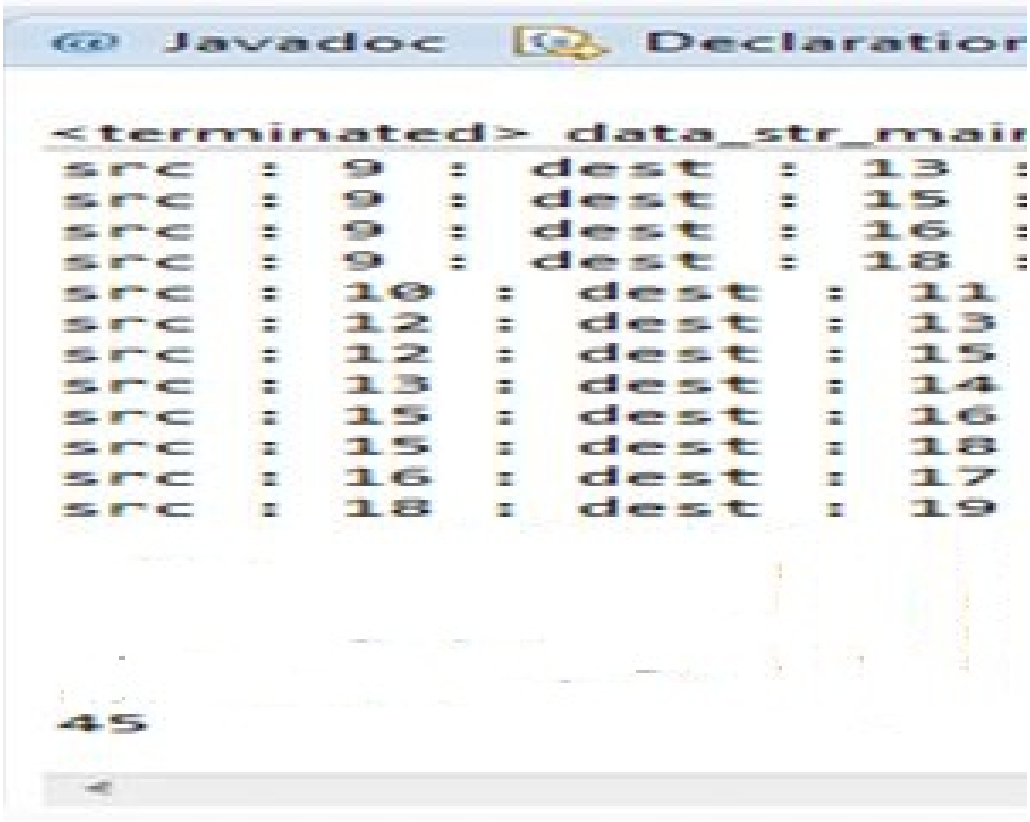

Figure 4.8: Time required for slicing (t2)=45ms.

Total time =Time for SDG generation + Time for slicing= t1+t2 =187+45 =232ms.

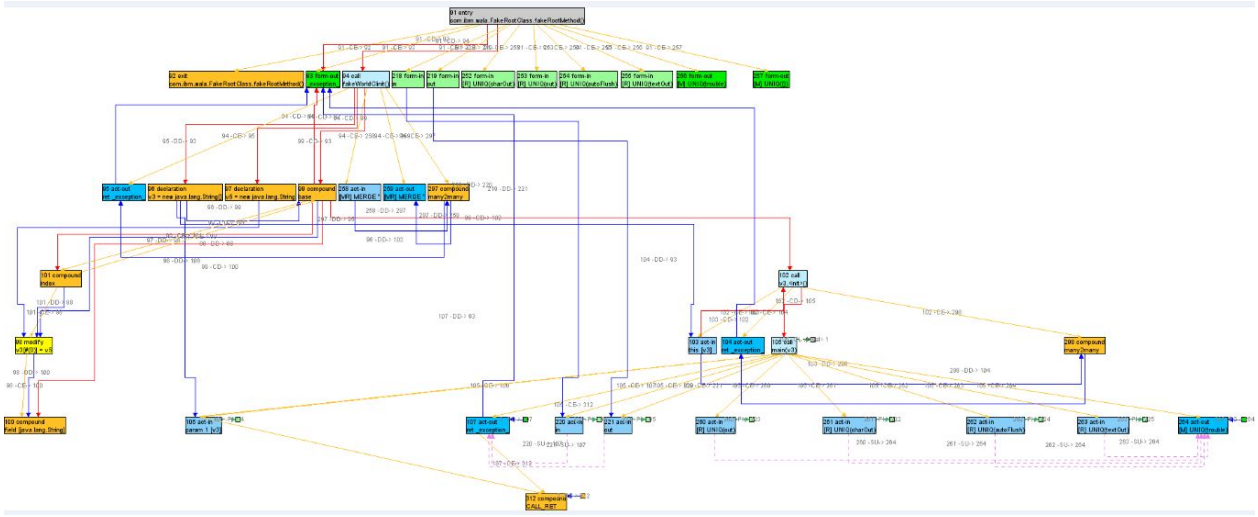

Figure 4.9: A System Dependence graph of the program shown in Figure 4.3

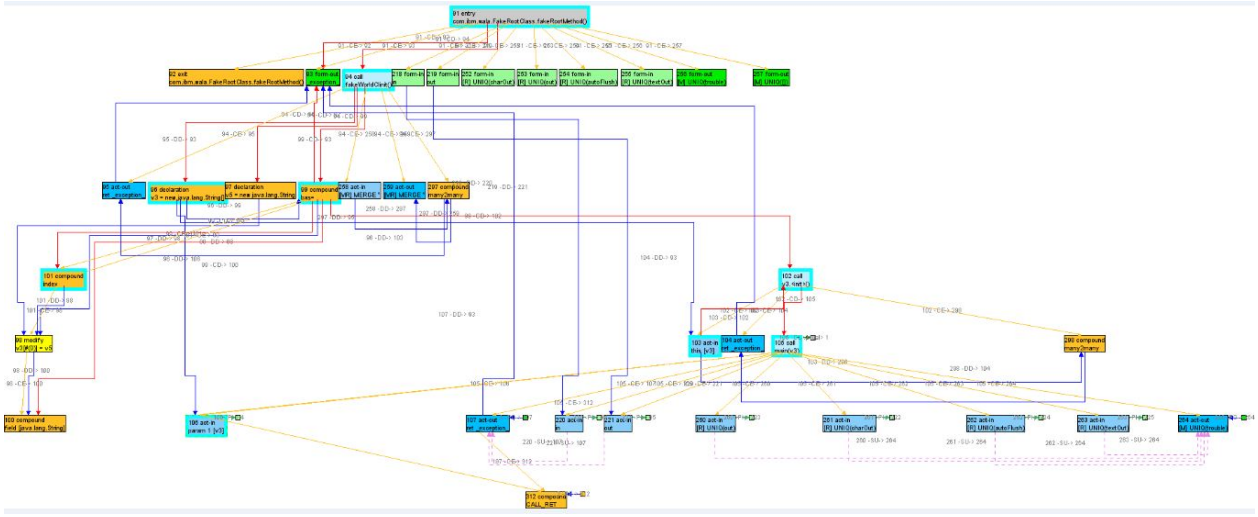

Figure 4.10: Sliced System Dependence Graph.

```
> buildSDG
Setting up analysis scope... (from jar stream jSDG-stubs-jre1.4.jar) done.
Creating class hierarchy... (557 classes) done.
Setting up entrypoint FindLargest.main([Ljava/lang/String;)V... done.
Building system dependence graph...
        callgraph: 5 nodes and 8 edges
        intraproc: calls.mergeable..clinit.statics.heap(if,adj,df,reg).misc.killdef.convert.summa
done.
Time needed: 187ms - Memory: 105M used.
```

Figure 4.11: Time required for SDG generation=187ms.

```
package hello;

public class hello {
    public static void main(String args[])
    {
        System.out.println("Hello");
    }
    }
```

Figure 4.12: A Sample Source Program.

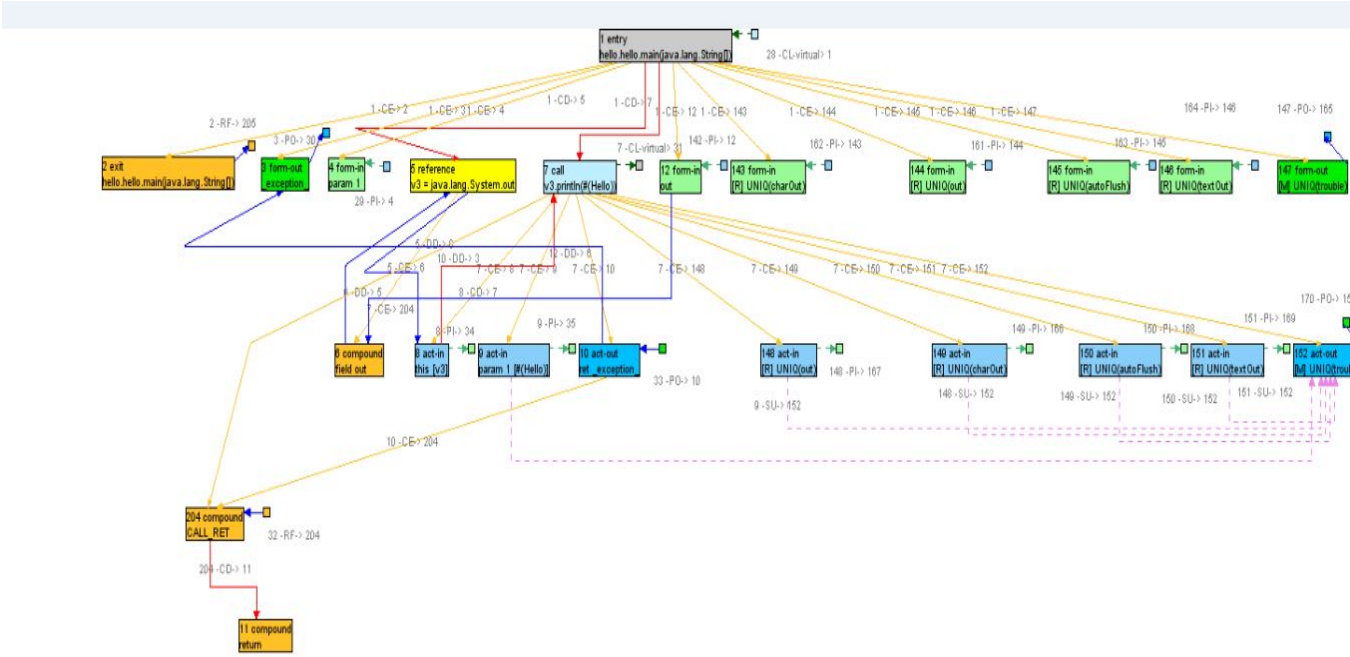

Figure 4.13: A System Dependence graph of the program shown in Figure 4.12.

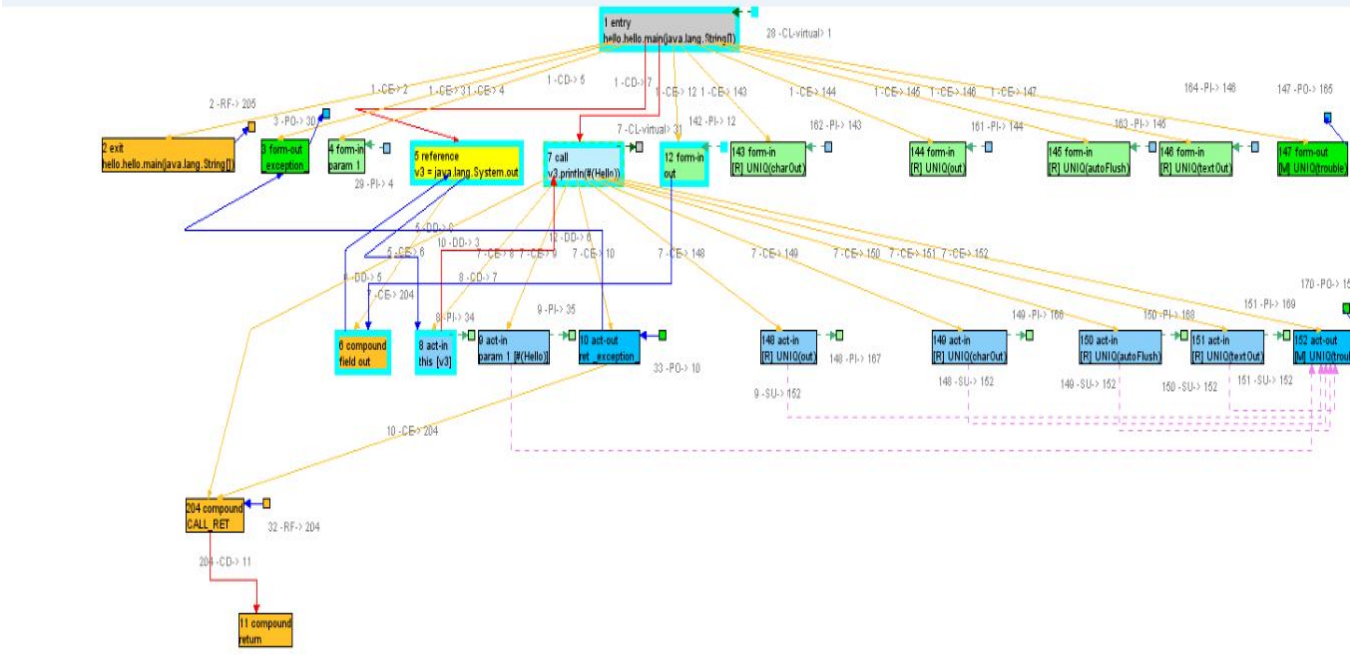

Figure 4.14: Sliced System Dependence Graph (Slice generated by performing slicing at node 7.)

```
> buildSDG
Setting up analysis scope... (from jar stream jSDG-stubs-jre1.4.jar) done.
Creating class hierarchy... (553 classes) done.
Setting up entrypoint hello.hello.main([Ljava/lang/String;)V... done.
Building system dependence graph...
        callgraph: 5 nodes and 4 edges
        intraproc: calls.mergeable..clinit.statics.heap(if,adj,df,reg).misc.killdef.convert.summa
done.
Time needed: 166ms - Memory: 50M used.
```

Figure 4.15: Shows time required for SDG generation using Joana for sample program shown in Figure 4.12.

Chapter 5

Comparative analysis between Joana and Java SDG API

Table 5.1 shows that both the frameworks support the analysis of Java bytecode and generation of SDG. The different features that are supported by either of the frameworks are also shown in Table 5.1. There are many features that each one of the framework supports. Here, we have considered each one of these features to perform a comparative study. Figure 5.1 shows a comparative study between Joana and Java SDG API frameworks by taking eleven different programs into consideration. Also, we have taken a comparison between Joana and Java SDG API in terms of slice computation time which is shown in Table 5.2 . This is evident that Java SDG API

Table 5.1: Comparing Java SDG API and Joana.

Framework	Joana	Java SDG API
SDG Generation	Yes	Yes
Slicing	Yes	Yes
Chopping [9]	Yes	No
Integrity and confidentiality	Yes	No

is a more prominent framework in terms of nodes and edges as it requires less number of nodes and edges than Joana framework. Thus, requiring less space to represent the intermediate SDG. As a result Java SDG API seems to be more scalable for industrial applications. On the other hand, Joana is more efficient in generating the required SDG in lesser time. However, the accuracy of the intermediate graph generated by Java SDG API is essential to be studied and is left for future work.

Table 5.2: A Comparison of slicing time between JOANA and Java SDG API based on the input programs.

S.No	Program	LOC	Slicing criterion: (L, v)	Time required by Java SDG API for slicing(ms)	Time required by Joana for slicing(ms)
1	Find largest number	23	<19,z >	45	37
2	Binary Search	38	<25,middle >	73	51
3	Quick Sort	49	<45,i >	129	112
4	Check Palindrome	22	<22,temp >	38	34
5	system clock	19	<21, day >	31	26
6	Type casting	26	<25, j >	36	28
7	Factorial	20	<18, output >	37	31
8	Fibonacci series	22	<20, j>	35	30
9	Floyds triangle	19	<16, k>	34	29
10	Armstrong	24	<24, temp>	40	34
11	Decimal to binary conversion	20	<20, m>	34	30

The contents of Table 5.2 and 5.3 is represented in the form of bar chart in Figure 5.1. In Figure 5.1, X axis represents program names and Y axis represents the time required for SDG generation and slice computation. That bar chart shows that Joana is more effective in terms of time required for computing slices.

Table 5.3: A Comparison of SDG generation time using JOANA and Java SDG API based on the input programs.

S.NO.	Program Name	Joana			Java SDG API		
		Number of nodes	Number of edges	Time required for SDG generation(ms)	Number of nodes	Number of edges	Time required for SDG generation(ms)
1.	Find largest number	264	1374	187	19	41	187
2.	Binary Search	546	2921	225	29	64	722
3.	Quick Sort	443	2391	208	63	119	458
4.	Check Palindrome	843	4521	268	17	31	457
5.	Demonstrate system clock	973	5408	304	13	25	388
6.	Demonstrate type casting	1689	9722	406	16	24	409
7.	Factorial using recursion	378	1934	187	22	31	406
8.	Fibonacci series	744	4039	261	19	28	407
9.	Floyd's triangle	927	5002	285	18	28	399
10.	Check for Armstrong	897	4866	284	18	33	442
11.	Decimal to binary conversion	825	4407	287	16	27	388

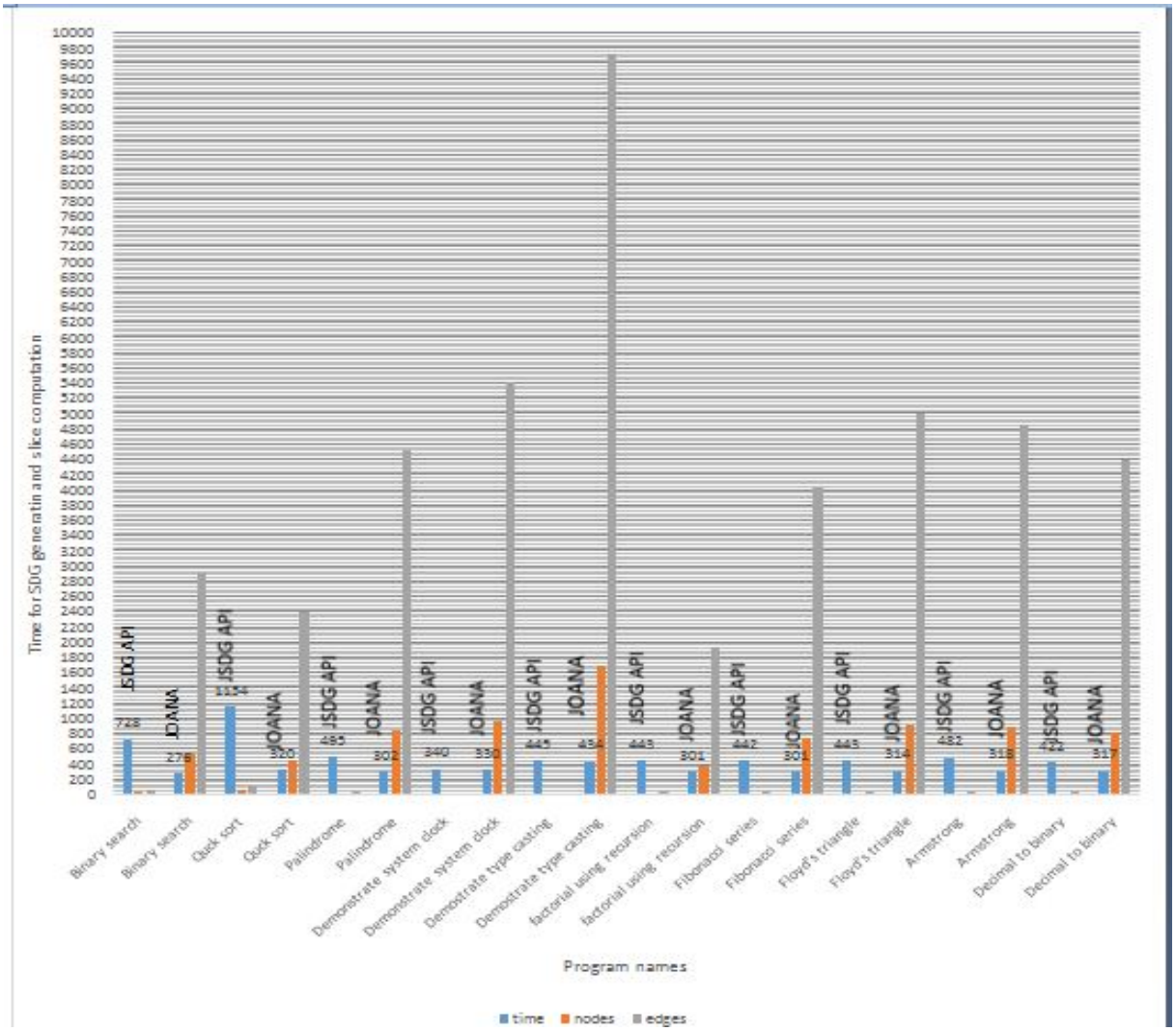

Figure 5.1: Bar chart showing the timing analysis of Joana and Java SDG API.

Chapter 6

Conclusion and Further work

The existing literature shows that program slicing concept helps in solving different types of problems. A generalization or combination of earlier slicing technique is required by every applications. In our work, we have reified the SDG generation and computed slice by applying backward slicing approach using Java SDG API and JOANA frameworks. Also, we have performed a comparative analysis between these two frameworks by taking various parameters into consideration such as number of nodes, number of edges and the time of computing the slices. The comparative analysis study shows that Joana provides more features than the Java SDG API and also it is more efficient with respect to the time required to generate the SDG. Whereas, in terms of number of nodes and edges, Joana requires more number of nodes and edges. Hence, the space complexity of Java SDG API is better as compared to Joana.

As for the future work, we are planning to utilize the Java SDG API framework for providing integrity and confidentiality to the information within the program and also perform chopping using Joana for program testing and security analysis. We also focus to take some industrial benchmark programs to carry out our analysis.

Dissemination

1. Ranjan Kumar, Subhrakanta Panda and Durga Prasad Mohapatra, *Analysis of Java Programs using Joana and Java SDG API*, 4th international conference on Advances in computing, communication and Informatics (ICACCI-2015). 10-13 August, Kochi, Kerala, India.(Communicated)

Bibliography

[1] T.Wang, A.Roychoudhary, "Slicing on Java Bytecode trace", ACM Trans. Program.syst. 30, 2, Article 10, March 2010.

[2] N. Walkinshaw, M. Roper and M. Wood,"The Java System Dependence Graph", in the Proceedings of the third IEEE international workshop on sourcecode analysis and manipulation, pp. 55-64, September 2003.

[3] The IFC(Information flow control) console and Graph Viewer, http://pp.ipd.kit.edu/projects/joana/.

[4] F. Umemori, K. Konda, R. Yokomori and K. Inoue, "Design and implementation of bytecode-based java slicing system", in Proceedings of the third IEEE International Workshop on Source Code Analysis and Manipulation, pp. 108-117, 2003.

[5] A JSDG(Java System Dependence Graph) API, http://www4.comp.polyu.edu.hk/ cscllo/teaching/SDGAPI/.

[6] M. Weiser, "Program Slicing", IEEE Transaction on Software Engineering, pp. 352-357, 1984.

[7] J. Silva, "A vocabulary of program slicing-based techniques", ACM computing surveys(CSUR), 44(3), June 2012.

[8] E. Kuleshov, "Using the ASM framework to implement common java bytecode transformation patterns", Aspect-Oriented Software Development, 2007.

[9] J. Krinke, "Barrier slicing and chopping",in Proceedings of third IEEE International Workshop on Source Code Analysis and Manipulation, pp. 81-87, September 2003.

[10] D. Liang and M. J. Harrold, "Slicing objects using system dependence graphs", in Proceedings of international conference on Software Maintenance, pp. 358-367, November 1998.

[11] H. Agrawal and J. Horgan, "Dynamic Program Slicing", SIGPLAN Notices, vol. 25, no. 6, pp. 246-256, 1990.

[12] Z. Ujhelyi, A. Horvth and D. Varr, "Dynamic backward slicing of model transformations",IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 1-10, April 2012.

[13] L. Chen, and B. Xu, "Slicing Java generic programs using generic system dependence graph", Wuhan University Journal of Natural Sciences, 14(4), pp. 304-308, 2009.

[14] B. Xu, J. Qian, X. Zhang, Z. Wu and L. Chen, "A brief survey of program slicing", ACM SIGSOFT Software Engineering Notes, vol. 30, no. 2, 2005.

[15] De Lucia, Andrea, "Program slicing: Methods and applications", In 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 0144-0144. IEEE Computer Society, 2001.

[16] J.Zhao, "Slicing concurrent Java programs", In Proceedings of Seventh International Workshop on Program Comprehension, pp. 126-133, 1999.

[17] J. Ellson, E. Gansner, L. Koutsofios, S. C. North and G. Woodhull, "Graphvizopen source graph drawing tools", In Graph Drawing, pp. 483-484. Springer Berlin Heidelberg, 2002.

[18] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs", ACM Transactions on Programming Languages and Systems (TOPLAS), vol.12, no. 1, pp.26-60, 1990.